Practical Work on the Project session 1
Moving map

This work session is dedicated to learning/revising the important things to develop the graphical components for a static map in a first time, then a moving map showing the traffic (GPS-like). **You will need to finish the *OOP PW3* and *4* before this session** in order to move forward on this topic. **duration: 2h**

In the *OOP PW3* we retrieved the airspace description from files. Then we displayed them in textual form in the standard output, to roughly test them. Now we are going to create the graphical components to display them as a map.

The airspace description files provide coordinates in *lat/lon* format, which is not a Cartesian coordinate system, and therefore cannot be displayed directly on a screen.

The Cartesian coordinate system used by the French ATC services is called *Cautra*. It's a stereographic projection of *lat/lon* coordinates around a specific point located somewhere in France mainland. Once it's y-axis is inverted it can be displayed directly (it is oriented upwards as in standard coordinate systems, whereas it is oriented downwards in most graphic libraries, including *JavaFX*). The *Rejeu* traffic simulator you have already used (*OOP PW4*) returns the coordinates in *Cautra* format.

The easiest way to display the points coming from both origins in a common coordinate system is therefore:

- For *Cautra* format, to invert the y-axis,
- For *lat/lon* format, to convert into a *Cautra* with the y-axis inverted format. For this last case, the `cartoxanthane` library performs this operation for you when loading the data (it's free!) and delivers directly displayable points.

Exercise 1: Draw the base map view (global coordinates)

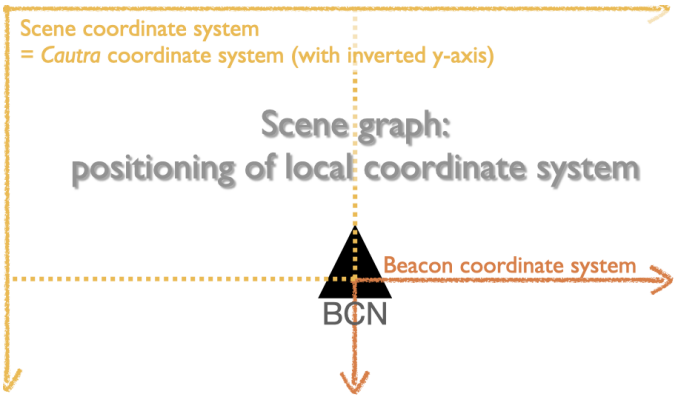
If you have finished the *OOP PW3* correctly you should have already used in your model an instance of the class `BaseMapXanthane`, implementing the interface `IBaseMap`, that describes the french base map. This class contains a list of objects implementing the interface `IZone` (here class `OutlineXanthane`) which describe closed zones like the mainland or an island. These `IZone` are themselves constituted of a list of objects implementing the interface `IPoint` (here class `VertexXanthane`). The goal of this exercise is to draw the view associated to this model:

- Create a class `BaseMapView` extending a *JavaFX* `Pane`,
- Look into the documentation of the *JavaFX* class `Polygon` and create as many polygons as there are `IZone` in your `BaseMapXanthane` instance.

Exercise 2: Draw the beacon view (local coordinates)

With the same logic, create the class `BeaconView` which is the view associated to a beacon model, specified by the interface `IBeacon` (here class `BeaconXanthane`). Here are a few indications:

- In a first time just extend `Pane` (hint: this will change depending on the interactions that will be added later. If you guess it, inherit directly from the right type),
- The beacon view is richer than the base map view (it's composed of several types of objects, and will evolve when we add moving map features and interactivity), so this time you'd rather use local coordinates as this was done for the `TranslatableHomotheticPaneRect` in *RGIP PW3*: the rectangle was located in the local coordinate system at (0,0). Then the whole component was placed at a specific location inside the scene and was manipulated as a whole. In our case the triangle must be placed around the origin of your component `BeaconView` that will then be located in its parent, at the beacon position memorized in the model (see graph below).



- There is no class Triangle in *JavaFX*, use `Polygon` here again,
- Place a `Text` instance below, to display the beacon name. As we intelligently use the local coordinate system, this text widget, like any of the other `BeaconView` subcomponents, will just be placed relative to the triangle (no need to take the beacon's location into account). Make sure that the text widget is horizontally centered with the triangle (invoke the method `getBoundsInLocal()` on the text in order to get its size and compute its abscissa).

From now on, keep on thinking local for all the other components. It will help you to easily place subcomponents and apply geometric transforms.

Exercise 3: Static map displaying the drone

3.1 Static map:

- Create a class `RadarView` displaying the complete map (inheriting from `Pane`),
- Create a graphical component `SelfTrafficMarker` to simply mark the position of your drone (so for example a small circle in a `Pane` for the moment),
- Add an instance of `SelfTrafficMarker` in the class `RadarView`.

Now you have to make it move with your drone…

3.2 Drone position update:

If you have finished the *OOP PW4* correctly you now know how to retrieve the position of your drone with *Rejeu*. Take the code you produced and integrate it into your project.

Also, if you remember the discussions in *RGIP PW1* and *2* about the separation between model and view, you now know that the position (and other characteristics) of your drone must be stored in a specific model (outside the `CommunicationManager`), which will be observed (at least) by a view (`SelfTrafficMarker`):

- Create the selfTraffic model class in a package named `model`,
- Update it regularly from your `CommunicationManager` with the data received from Rejeu,
- Update the position of the view by using *JavaFX*'s observability mechanisms.

You now have a graphical component displaying the moving drone on a static map background. The next step will be to implement a GPS-like moving map: a map whose position will be centered on the drone's position and which will be oriented so that the drone's heading points upwards…

3.3 Interaction:

Which of the direct manipulation interactions you learned to code in *RGIP PW3* for a static map can be implemented for a moving map, and which should not? For what reasons?

What are the architectural implications of these choices for your code (you've already had hints before)?

This question was just to get you thinking about the improvements we've made to the basic direct manipulation interactions, which you'll need to be inspired for the next exercise…

For now, don't change your code, just write down the things to do in TODO comments. This is really important to trace this kind of things in a project. And with *NetBeans* you can view these special comments (TODO, FIXME... or any custom syntax) by selecting the *Window* > *Action Items* menu.

Exercise 4: Moving map

4.1 What are the properties of the drone to observe in order to animate the moving map?

4.2 Design the scene graph with positioning and transformations in mind:

Imagine a branch of the scene graph of your future application allowing to place and transform (moving map feature + zoom interaction) in the simplest possible way the different graphical components you have just created. Draw your scene graph by indicating the name of the components in your code, their class, their position in the local coordinate system of their parent and the geometric transformations that must be applied to them to obtain your moving map.

This scene graph will have to be enriched as you work on the design of the project, used for coding, and delivered for documentation purposes.

4.3 Basic implementation:

Code the moving map according to your scene graph.

Code a simple visual test to check that the map stays centered on the drone's position.

4.4 The centering of the map on the drone's position works well, but its orientation according to the drone's heading has an obvious flaw. What is it?

How to fix it (think of a solution similar to the improvements we've made to the basic direct manipulation interactions)? Depending on the choice of local/global coordinates, the solution can be very simple to implement or not… Why?

Fix it!

You are now really ready to start designing and coding the GUI of your project, starting with the Navigation Display-like component of your *ground system*.

Exercise 5: First steps towards Navigation Display

5.1 Following the design guidance from the previous exercise, now imagine the missing graphical components to create the Navigation Display. Update the scene graph.

5.1 Compass layout:

The compass is the graduated circle in the Navigation Display indicating the heading. Describe a simple (think local!) and reusable (not a big single class!) way to draw it and update the scene graph.

5.2 Compass update:

Describe a simple way to update the compass in response to the drone's heading.

Now it’s time to code! You will not have another tutored session on the Navigation Display, so remember to properly document your design (UML, scene graph, TODOs/FIXMEs, javadoc… and all the other courses deliverables), to use the whole as a rigorous basis for coding, and for asking questions if you have problems.